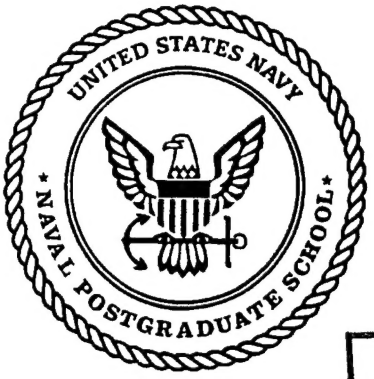
NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

CODE INSPECTION FOR NPSNET

by

Charles E. Adams

March 1995

Thesis Co-Advisors: Timothy J. Shimeall
John S. Falby

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Master's Thesis

4. TITLE AND SUBTITLE
CODE INSPECTION FOR NPSNET

5. FUNDING NUMBERS

6. AUTHOR(S)
Adams, Charles, E.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The virtual reality research at the Naval Postgraduate School has produced a simulation environment called NPSNET. NPSNET demonstrates that a real-time, interactive three-dimensional simulation system for multiple networked participants is achievable using low-cost workstations. However, as NPSNET expands, limitations of the current testing methods have become apparent, particularly in the area of man-hours spent in detecting faults in the software. The problem addressed by this research was to improve the validation process of NPSNET by implementing an efficient code inspection.

The approach taken was to develop a two-person code inspection based on Fagan's Inspections. The development of the inspection process began with the inspection checklist. The checklist is a result of studying the software development difficulties of NPSNET and other code inspection checklists. Next was the design of the inspection process, which focused on streamlining the amount of time and number of participants conducting the inspection. Finally, a trial inspection was conducted to provide feedback on the effectiveness of the software inspection process.

The results of this work demonstrate that it is possible to develop a fast and effective inspection process with fewer people required to conduct it. The trial inspection reduced the time from four to two hours to complete, produced a 35% defects per lines of code rate, and only required two instead of four people to conduct.

14. SUBJECT TERMS	15. NUMBER OF PAGES
NPSNET, code inspection checklist and inspection process.	60
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

CODE INSPECTION FOR NPSNET

Charles E. Adams
Lieutenant, United States Navy
B.S.E.E., University Of California at Davis, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1995

Author: Charles E. Adams
Charles E. Adams

Approved by: Timothy J. Shimeall
Timothy J. Shimeall, Thesis Co-Advisor

John A. Falby
John S. Falby, Thesis Co-Advisor

Ted Lewis
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The virtual reality research at the Naval Postgraduate School has produced a simulation environment called NPSNET. NPSNET demonstrates that a real-time, interactive three-dimensional simulation system for multiple networked participants is achievable using low-cost workstations. However, as NPSNET expands, limitations of the current testing methods have become apparent, particularly in the area of man-hours spent in detecting faults in the software. The problem addressed by this research was to improve the validation process of NPSNET by implementing an efficient code inspection.

The approach taken was to develop a two-person code inspection based on Fagan's Inspections. The development of the inspection process began with the inspection checklist. The checklist is a result of studying the software development difficulties of NPSNET and other code inspection checklists. Next was the design of the inspection process, which focused on streamlining the amount of time and number of participants conducting the inspection. Finally, a trial inspection was conducted to provide feedback on the effectiveness of the software inspection process.

The results of this work demonstrate that it is possible to develop a fast and effective inspection process with fewer people required to conduct it. The trial inspection reduced the time from four to two hours to complete, produced a 35% defects per lines of code rate, and only required two instead of four people to conduct.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

A thesis is never the work of one individual. There are several people who contribute to this work both directly and indirectly. I would like to express my sincere thanks to the many people who helped me prepare and implement this thesis. In particular, I would like to take this opportunity to thank Dr. Timothy J. Shimeall, for his expertise, guidance and support from the birth of this idea to its fruition. Secondly, John S. Falby whose knowledge and support assisted in making this thesis precise and fluent. A special thanks goes out to my wife Darla for her patience, encouragement and loving support throughout this process. Finally, I would like to thank my parents David and Harriet Adams for their love, inspiration and leadership. Without the help and moral support of the above people and others, this work would not have been possible.

I. INTRODUCTION

Three-dimensional interactive visual simulation systems, known as virtual reality systems, have become an increasingly preferred mechanism for the development of complex systems for the Department of Defense and for operator training for those systems. This preference is because virtual reality systems save on training cost, training time and, most importantly, human lives.

If virtual reality systems are to be used by the Department of Defense with human lives at stake, the results of the simulations must be trusted. One aspect of such trust is trusting the reliability of the software that produces such results. Validation of a virtual reality system is one form of establishing confidence and building trust into these systems.

To aid in the validation process, a system of testing criteria that takes advantage of both static and dynamic testing needs to be incorporated in the software development process. This technique exploits both human and computer strengths in the testing process. The human strengths are exploited by directing the evaluation to important portions of the design and source code in an execution-independent static fashion. Computer strengths are exploited by standard testing techniques, which are long and exhausting tasks that humans do not perform well. By exploiting the strong points of both, significant improvements can be made in the validation of virtual reality systems based on their successful use on other software systems, and their complementary error detection [Ref. 1].

A. NPSNET

The virtual reality research at the Naval Postgraduate School has produced a simulation environment called Naval Postgraduate School Networked Vehicle Simulator (NPSNET) [Ref. 2]. NPSNET demonstrates that a real time, interactive three-dimensional simulation system for multiple networked participants utilizing Simulation Network (SIMNET) databases and SIMNET and Distributed Interactive Simulation (DIS) formatted networking is achievable using low-cost workstations. NPSNET models both real-world

and special-purpose terrain databases as well as a variety of imaginary and real vehicles, including tanks, aircraft and helicopters. The user can operate any vehicle not currently being used by another user and can fire on any other vehicle. A user's vehicle can interact in real time with the vehicle of any other user on the network as well as autonomous vehicles controlled by the system or other workstations.

However, as NPSNET expands, limitations of current testing techniques have become apparent, particularly in the area of detecting faults in communication-control software. In addition to NPSNET, improved testing techniques may aid other real time projects at the Naval Postgraduate School, such as the Autonomous Underwater Vehicle [Ref. 3] and the Yamabico autonomous robot [Ref. 4], which have been plagued by similar limitations of current testing techniques.

B. REVIEWS, WALKTHROUGHS, AND INSPECTIONS

Research testing of software in the Naval Postgraduate School NPSNET research group has been accomplished primarily through computer-based testing techniques. Non-computer-based testing ("human testing") needs to be explored as a supplement to current NPSNET research testing methods. Experience has shown that these "human testing" techniques are quite effective in finding errors. Common recommendations are that one or more of these should be employed in every programming project [Ref. 5].

Reviews, walkthroughs, and inspections are the primary "human testing" methods. All involve the reading or visual inspection of a portion of the project (project element) by a team of people, with one goal in mind: to find errors [Ref. 5]. The differences between them are subtle but distinct and are presented here based on the IEEE Standards Board [Ref. 6].

The objective of a technical review is to examine a project element and provide management with evidence that: the project element conforms to specifications; the development of the project element is according to plans, standards, and guidelines; and changes to the project element are correct and only affect areas specified in the change

specification. The recommended number of participants in the technical review team is three or more people depending on the size and complexity of the software element being examined. The review team comprises members from technical leadership and a variety of occupational positions. No formal data collection is required nor is a database maintained for trend analysis.

The objectives of a walkthrough are: to find defects, omissions, and contradictions in the project element; to improve the project element; to consider alternative implementations; to exchange techniques and style variations; and to educate the participants. The group size is between two to seven people represented by technical leadership and a variety of occupational positions. No formal data collection is required nor is a database maintained for trend analysis.

The objective of a software inspection is to detect and identify project element defects, with the purpose of: verifying that the project element conforms to specifications and standards, identifying deviations from specifications and standards; and collecting software engineering data. Software inspections do not explore alternatives or stylistic issues. The group is composed of software engineers with the size being from three to six persons. Data collection is formally required with database entries on items such as defect counts, error characteristics, and severity, for trend analysis and planning.

One could view the differences between reviews, walkthroughs and inspections from a presentation perspective. Reviews are in general a corporate presentation of the product to the key decision makers. Walkthroughs are on a smaller scale and are designed to examine alternatives and be a forum for learning. Software inspections are designed to be a small meeting of the minds on the product vice a presentation of the product.

The Department of Defense (DoD) policy on formal reviews is based on DOD-STD-2167A, which states: "During the software development process, the contractor shall conduct or support formal reviews and audits as required by the contract" [Ref. 7]. The DoD technical review process is remarkably similar to the IEEE standard. Specific guidance on formal reviews by the DoD specifies that they will be done, when they should

occur, and what documents will be reviewed and produced by the contractor. Further guidance on formal reviews is provided in MIL-STD-1521B [Ref. 8].

C. RESEARCH QUESTIONS

As stated earlier, software testing of NPSNET has been accomplished primarily utilizing computer-based testing techniques. A designer runs test cases on code that they have developed and the research group runs the current testing suites on the product with the new code implemented. Utilizing these techniques alone has significantly reduced productivity in coding, because errors were not found early enough in the software development process increasing the amount of rework time to correct problems [Ref. 9]. The use of inspections could improve productivity in coding by detecting errors earlier.

The process of developing an inspection process started with a study of the development process and product difficulties of NPSNET. These difficulties then become the framework for development of the inspection process and inspection materials.

How can inspection techniques be adapted to university research efforts? The process of design and code inspections in their present form requires four to six participants and takes on the average about four to five hours for each participant. Time and people go hand in hand, as people have other commitments (i.e., courses, projects, homework, meetings, etc.) which limits the amount of time for research, let alone inspections. Given the constraints on time and people (lack of), one needs to tailor inspections to minimize the amount of time and people used to conduct them.

D. SUMMARY OF CHAPTERS

Chapter II presents the purpose of the checklist, its development and use, followed by a discussion of the inspection process. Chapter III discusses the trial inspection from module selection to analysis of results. Chapter IV summarizes this thesis and its results. It also explains thesis application and suggestions for future research. The initial and final NPSNET code inspection checklists are contained in Appendix A and Appendix B respectively.

II. CODE INSPECTION CHECKLIST

Checklists are a fundamental part of the software inspection process. There are individual checklists for each type of document reviewed. Checklists are needed for the inspection process alone, and are normally not used in the initial production of the product. An analytical tool in the inspection process, checklists need to be available to the inspectors for individual checking, because this is when they are used the most. Tom Gilb defines the inspection checklist as: "A specialized set of questions designed to help checkers find more defects, and in particular, more significant defects. Checklists concentrate on major defects." [Ref. 1]

A. PURPOSE

The primary goals of the checklist are to instruct, stimulate and increase performance of the inspection team. Checklists provide guidance to the inspection team on what is expected. By having a list of questions to refer to, the novice inspector has an idea of what to look for (i.e., a recipe for finding defects). This allows the novice inspector to be an effective member of the inspection team with little or no prior training.

Organized properly, checklists can provoke the inspection team to look for more than they might otherwise. With the checklist covering a variety of areas, it prevents the inspection team from concentrating on only a couple of areas. It also provides a valuable memory aid for the experienced inspector, stimulating more questions drawn from past experiences.

Ultimately, checklists measurably increase the number of defects found by the inspection team. Because they are based on the most common defects made by developers, checklists focus the attention of the inspection team on those areas where defects are most likely to occur. Therefore, they increase the probability of detection by the inspection team.

B. DEVELOPING CHECKLISTS

Checklists should be based on experience developed locally, and should not be copied from other environments. Experience has shown that copied checklists do not actually contribute to the identification of major defects. They are often either obvious or irrelevant, and are usually an attempt to enforce one's own standards. [Ref. 1]

By developing the checklist locally, based on one useful question at a time, the checklist becomes tailored to one's own needs. The stimulus for this is when it is clear that a major defect has been identified by an inspector who has asked the question of the document that others could have asked, but did not. One needs to capture that insightfulness in the form of a checklist question. The inspection leader must be trained to recognize this situation and to take full advantage of it by asking the inspector to identify the key question or analytical process used in finding that defect. [Ref. 1]

Checklists should be validated periodically. A simple test to check the validity of checklist questions is to log the question's identification tag and generate data showing which checklist questions are resulting in defects detected. If some questions are not registered over a reasonably long period, then those defects are no longer a common source for error and those questions should probably be deleted from the checklist. [Ref. 1]

Checklists should be constantly evolving. As producers of the project become accustom to the checklist, old defects are resolved which give rise to new defects which will require new checklist questions to detect them. This will become more apparent to the inspection team over time with use and from results of periodic checklist validations. Anytime a major defect has been identified by anyone that is not covered in the checklist is proper cause for inclusion in, and updating of, the checklist. [Ref. 1]

C. NPSNET CODE INSPECTION CHECKLIST

1. Checklist Development

The process of developing the NPSNET code inspection checklist began with the primary programming language NPSNET is written in, C++. I conducted a search of

existing checklists or documented literature on common sources of programmer errors in C++. The initial NPSNET checklist is based on the checklist developed by John T. Baldwin, "An Abbreviated C++ Code Inspection Checklist" [Ref. 10] and the book by Scott Meyers from his experiences from teaching, "Effective C++" [Ref. 11]. I conducted an additional search on common sources of errors inherent to programmers, not language specific. Questions from the checklist by Glenford J. Myers, "The Art of Software Testing" [Ref. 5], were also incorporated into the NPSNET checklist.

To tailor the initial checklist to NPSNET, I conducted interviews with the NPSNET research group to incorporate questions specifically related to NPSNET difficulties. Questions that arose from these discussions were language specific pertaining to students' understanding of C++. The NPSNET research group expressed an interest in addressing style issues in the checklist. A separate section of the checklist incorporates these issues based on John Falby's document, "Style Guide for Winter AY95" [Ref. 12]. In the discussions, graphical issues raised were based on performance with the misuse of types in computations with IRIS Performer. "IRIS Performer is an application development environment that combines a programming interface for creating visual simulation applications and a high-performance rendering library in one easy-to-use 3-D software toolkit. IRIS Performer provides a flexible, intuitive, toolkit-based solution for developers who want to optimize performance on Silicon Graphics systems" [Ref. 13]. With that preference in mind, I merged questions into the checklist to address performance optimization.

2. Initial NPSNET Checklist

The checklist is laid out in terms of category blocks. Within each block there are category sub-items, and each sub-item represents a checklist question. This organization allows users to easily locate a set of questions associated with a particular block of code during the inspection. It also allows one to efficiently maintain the checklist as it undergoes revisions from the removal and addition of old and new checklist questions. The categories

are chosen based on programming issues, C++ language constructs, and performance issues with IRIS Performer.

The programming issues addressed in the checklist cover data declarations, data references, local and global variables, allocating and deallocating data, casting, files, computations, comparisons, conditionals, control flow and variables, branching, interfaces, and argument passing. These comprise generic questions about good software engineering practices, in general, that do not fall into the specific language construct categories (i.e., Arrays, Constants, Classes, etc.).

The C++ language constructs presented are arrays, constants, classes, strings, pointers, assignment operator, and functions. These categories deal with language specific details and established standards on proper usage to avoid common errors in programming.

Performance issues that are specific to IRIS Performer have been incorporated into the previous mentioned categories for clarity in usage of the checklist. For example, to take full advantage of the IRIS Performer math package, one should avoid the use of short integers and doubles in computations to increase performance. Instead of including a separate category to handle this, such performance questions were added to the appropriate variable declaration categories.

Once developed, the initial NPSNET checklist (see Appendix A) was used on actual NPSNET source code. Based on lessons learned, the checklist has undergone revisions to its current state (see Appendix B).

3. NPSNET Logging Sheet

Errors found during inspection need to be accurately and efficiently recorded. The checklist is set up with each checklist question having a unique identification tag made up of a number representing the category and a alphabetic character associated with each category question. Example: checklist ID tag = 1A, means category **1** Data Declaration and the **A** represents the first question, "Default attributes understood (i.e., no lazy declarations)?" Using the checklist identification tag simplifies the logging process

considerably, thus allowing more time for inspection and error detection. The format for logging errors on the logging sheet is by source code line number, checklist identification tag, and explanation of error (optional). If an error is detected that is not addressed as a checklist question, one signifies it by the symbol "?" in the checklist identification tag field and annotates the defect in the explanation of error field on the logging sheet.

4. Use of the NPSNET Checklist

a. Producer

The producer of the code to be inspected should develop the code first without reference to the checklist. The purpose of the checklist is to trigger identification of issues and is not intended to direct the producer [Ref. 1]. Once the code has been generated, the producer should ensure that the code is as good as one can make it prior to the inspection. This means conducting one's own inspection first, ensuring the code meets the current entry criteria. At a minimum, the code should conform to the standards set forth in the style guide [Ref. 12]. This involves changing roles: the producer has to step out of the role of producer and take on the role of the inspector. The producer will conduct at least two inspections: one prior to, and one during, the inspection process. It is a personal challenge to the producer to see if they can find additional issues they might have missed in the pre-inspection.

b. Inspector

In terms of the inspection process, the time spent during individual checking is one of the most important. One needs to spend enough time to be effective at locating issues. This entails studying the documents, going over the checklist, and comparing them against each other. The goal is to find as many issues as possible. The inspector tries to concentrate on major issues, issues that no one else would find easily, but because of their expertise and experience, they are found, preventing serious problems later on in development. To conduct the inspection, the code is examined line by line, with the goal of

fully comprehending what one is reading. After each line or block of code, the checklist is searched for questions that apply. For each applicable question, the inspector determines whether the answer is no. A response of "no" to any question is a probable defect and should be logged. The checklist is not a complete list of what to look for, so the inspector must be prepared to ask questions and locate issues not addressed by the checklist. One final note, during the inspection the inspector should look for issues in the supporting documents and checklist as well, but remember the main emphasis is on the inspection document.

D. THE INSPECTION PROCESS (TWO PERSON)

1. Preparation

To prepare for the inspection, the producer makes separate hardcopies of the source code and supporting documents for oneself and the inspection leader. The hardcopy of the code to be inspected should show the line count. The inspection should be limited to no more than 250 lines of source code, including comments and excluding whitespace. [Ref. 10]

2. Inspection Overview

During this meeting, the producer takes about 20 to 40 minutes explaining to the inspection leader the general design of the code. The goal is to cover the code's main design features and limit the meeting to as close to 20 minutes as possible without undercutting the explanations. The inspection leader is not allowed to ask questions because the code is suppose to answer them. The overview is designed to speed up the process of understanding in order to allow more time to search for issues. If the inspection leader is already familiar with the producer's code, the overview is optional and can be omitted by the inspection leader. [Ref. 10]

3. Individual inspection

Each inspector (producer and inspection leader) uses the checklist to accumulate as many issues as possible, covering between 70 to 120 lines per hour. This has been determined to be the optimal checking rate for issues found. This should be completed in a single, uninterrupted, session. [Ref. 10]

4. Logging Meeting

The logging meeting is held for three purposes. The first is to log the issues found in the individual inspections. Issues are not recorded as defects, for historical purposes, until the producer has had a chance to evaluate them first. Issues are logged as items (potential defects or questions of intent to the producer) during this meeting. At this point, no discussion is allowed. Questions of intent are answered at the conclusion of the meeting. The second purpose is to identify more issues during the meeting. This is set up by allowing more time for checking. The third purpose is to identify ways of improving the inspection process, which includes, but is not limited to, improvement suggestions to procedures, rules or the checklist. The person who controls the logging meeting is the inspection leader and the producer logs all items presented. The inspection leader will ensure the producer receives a copy of the logging sheet for rework and sends a copy to the librarian. [Ref. 1]

5. Follow Up

The inspection leader is responsible for ensuring all items logged have been reworked satisfactorily. The items classified as defects must be corrected by the producer. The correctness of the rework will be verified at a short review meeting or another inspection. [Ref. 10]

6. Record Keeping

Record keeping is essential to the inspection process because it provides invaluable feedback to all involved. In order to objectively track the success of, and effectively improve, the inspection process, metrics must be collected. The inspection leader should

deliver the logging sheet to the appropriate member of the research group responsible for managing the records, the librarian. The librarian is responsible for generating all forms, producing all by-products, and storing information needed for, or resulting from, the inspection process.

7. Inspection Process Improvement

The librarian is the initiator of inspection process improvement. Based on the results gathered from each inspection, the librarian proposes changes to the research group before changes are made. Possible changes to the process are to the checklist and logging sheet. All suggestion for improvements should be addressed to the librarian.

III. TRIAL INSPECTION

The trial inspection is needed to provide feedback on the software inspection process and to demonstrate the need for that process. The results of the inspection provide insight to improve the checklist, inspection process, and research on NPSNET. The inspector for this trial inspection had limited background on the project. This was done to test the efficiency and effectiveness of the inspection.

A. INSPECTION MODULE SELECTION

To evaluate the inspection process in a structured fashion, careful consideration was given to the selection of a source document to review. This served two purposes: (1) provided valuable feedback to the majority of producers for whom the inspection process was tailored; and (2) establish the time frame to develop code for NPSNET to the amount of time the majority of producers had to work under.

With these goals in mind, Paul Barham of the NPSNET research group decided that **environ.cc** would best be suited for the trial inspection. The determination was based on the goals stated above, and to minimize the need for a thorough understanding of the interaction between modules in NPSNET. The latter determination was to allow the inspector to concentrate on the module being inspected during the inspection.

B. PREPARATION

With the module for inspection chosen, supporting files required in order to conduct a thorough inspection were obtained from the NPSNET research group. Before beginning the inspection, the inspector familiarized oneself with the supporting documents to acquire an understanding of the constants, definitions, global variables, and constructs used in NPSNET. Once the overview of the supporting documents was complete, a detailed understanding of the functionality of the module to be inspected was required. This process was begun by reading the document **environ.cc** for the sole purpose of understanding what supporting documents would be needed for the inspection. In retrospect, this should have

been done first as it would have saved on time and effort spent studying files unrelated to **environ.cc**. This was determined by inspection of all names against their definitions, which resulted in only one name untraced, NUMNODES. With NUMNODES being a global constant to NPSNET, the cost of finding its definition outweighed the gain. Based on the use of NUMNODES, it posed no serious threat to the operability of the code as long as NUMNODES was initialized to a value greater than zero. The file that contained NUMNODES was not apparent from a preliminary search. For that reason, no additional supporting code documents were required for the inspection.

A second pass through the **environ.cc** document was conducted to gain a thorough understanding of how the module functioned. Areas of concentration were user-defined definitions, data structures, and function used in the environ.cc file. To enhance understanding of the data structures used, pictures were utilized to visualize the concepts. By tracing through examples, a detailed knowledge of the operation of the functions in the module to be inspected was obtained.

C. INSPECTION

The inspection process identified three items: (1) errors in the module **environ.cc**; (2) questions that arise from discrepancies in the checklist; and (3) questions that arise about NPSNET in general. These three items are geared to improve the code generated by the NPSNET research group, improve the checklist, tailor the checklist to NPSNET, and improve the inspection process, as a whole.

The inspection was conducted by examining each line of code against the checklist, logging code defects on the logging sheet and annotating any questions that arose during the inspection process on the appropriate sheets labeled "Checklist Questions" and "NPSNET Questions". With the length of the module containing over 400 lines of non-commented source code, the inspection was broken up into several two hours sessions to prevent diminished performance [Ref. 1]. The minimum break time between sessions was one hour, to allow for recuperation. During the inspection process, defects were identified

and marked as such based on usage being unclear or wrong, according to the inspector's interpretation of the code and issues raised from the checklist.

D. ANALYSIS

1. Time

The total time to conduct the inspection from familiarization with supporting documents to the actual document inspected was 23.5 hours. This does not include the time for gathering or developing inspection materials for the inspection.

Time spent in familiarization with supporting documents was 3 hours. A lot of this effort could have been prevented in a normal inspection, because the participants would have been members of the research group and would have already been familiar with most of the supporting documents reviewed. An additional 3 hours was spent finding out what supporting documents were actually needed for the inspection by looking over the inspection document. As a result of not being the producer of the inspection document, unnecessary documents were reviewed for the inspection increasing time and effort spent, as mentioned earlier (section B. Preparation). By having the actual producer of the inspection document gather the necessary supporting documents, this effort would have been streamlined.

Before the inspection, 5 hours was required to understand how **environ.cc** functioned, in order to increase the effectiveness of the inspection. Time spent in this phase was inflated for two reasons: (1) no overview by producer; and (2) inspector was not a member of the research team. With a good overview by the producer of the inspection document, less effort is consumed in understanding the data structures, and the intent or logic of the producer, allowing the inspectors more time to concentrate on finding errors. By not being a member of the research group, the inspector did not have a complete understanding of how this module interrelated with other modules without obtaining further knowledge about NPSNET (big picture issues). The time spent in that effort would

have been part of the producer's overview or covered in the research group discussions prior to the inspection taking place.

The inspection itself took 12.5 hours to complete, covering about 34 lines per hour of non-commented source code. The standard for the inspection is to cover about 120 lines per hour [Ref. 1]. I attribute the excessively slow line rate to the amount of errors detected during the inspection. There were 160 errors found in 450 lines of non-commented source code, for an error rate of 35%, twice as high as the industry standard [Ref. 1]. A majority of the errors detected could have been eliminated by the producer, if the checklist was available at the time of production to ensure appropriate entry criteria were met, prior to conducting the inspection. By ensuring the inspection document conforms to style and format issues prior to the inspection taking place, one can concentrate on finding more meaningful defects, instead of consuming valuable time logging defects the producer should have addressed.

In summary, the majority of the time (over 60%) was in understanding NPSNET with no prior training. By having the producer and research group members participate in the inspection process, the time spent on the process is minimized, because the understanding and knowledge of the product is already present. With appropriate comments added to the source code files, one's understanding of those files would improve immensely. Prior to conducting an inspection, the producer should ensure the inspection module meets certain minimum entry criteria to improve the effectiveness and efficiency of the inspection.

2. Defects Detected

As stated earlier, 450 lines of non-commented source code were inspected producing 160 errors detected, resulting in an error rate of 35%. A distribution of the errors

detected is listed in Table 1. The errors are laid out by the checklist identification tag, type

ID Tag	Type Error	Instances	% of Errors
2A	magic #	42	26
4A	integer declaration	33	21
1A	lazy declaration	22	14
New	magic #	18	11
New	format #	13	8
New	commented out code	10	6
3B	constant declaration	10	6
26B	magic #	6	4
New	vague ID	3	2
1D	conflicting variables	1	1
4B	character declaration	1	1
New	improper design	1	1

Table 1: Distribution of errors detected

of error, number of occurrences, and percentage of errors detected. Issues that were raised that do not have a checklist identification tag are represented with a brief explanation of the defect. The majority of errors detected resulted from the use of hard-coded constants, over 40%. The excessive use of these "magic numbers" in the code hindered comprehension of code functionality and contributed greatly to loss of clarity. The second greatest source of errors detected was declaring variables to be signed when they only take on positive values.

There is no reason to allow illegal values to be stored in one's variable legally (strong type checking). The third common source of errors detected was lazy declarations. This could hinder performance with IRIS Performer. By not declaring the precision of floating point numbers, IRIS Performer promotes all undeclared floating point numbers to double precision, instead of single precision, slowing down performance. Once the producers become familiar with the checklist, the over-abundance of the three previous error types should diminish.

The distribution of the defect density is shown in Table 2, with an average defect density of 10 defects per page and a 7.08 standard deviation. As one reviews Table 2, there is no apparent pattern associated with the defect density, except uniformity over the document reviewed. In comparing the errors made in the first eight pages (84) to the last eight pages (76), they were relatively the same. I conclude that the average defect density is an initial benchmark of the amount of errors one can expect to find on any page generated by this producer prior to conducting an inspection.

Also included in Table 2 are categories for non-commented source code lines per page and page content. Page content has six classifications (comments, declarations, controls, conditionals, calculations and data usage), determined by the dominant classification on the page. A classification of comments was given to pages with less than 10 lines of non-commented source code and those pages were excluded in the analysis of page content. A one-way analysis of variance using t-tests on the data presented in Table 2 was conducted. The results of three tests from the series are presented in Table 3. First, the defects were studied against the code lines per page, running tests based on a variety of code line groupings. The results in Table 3 show a weakly significant association, with the majority of the defects occurring between 20 to 29 code lines per page. Next, the defects were analyzed against the page content, resulting in no significant association. After grouping the classifications into two groups, data and control, a strongly significant association was formed (see Table 3). The data group contained declarations, data usage and calculations, while the control group contained conditionals and controls. The results

demonstrate that the majority of the defects found were in data manipulation vice program control. For completeness, the defects content was studied. This is determined by the dominant classification of defects occurring on a page, using the same rules to classify page content. The analysis of defects against defects content resulted in no significant association (see Table 3).

Page	Defects	Code Lines	Page Content
1	0	1	declarations
2	20	35	declarations
3	11	24	declarations
4	11	27	controls
5	13	29	data usage
6	9	29	conditionals
7	16	37	data usage
8	4	37	conditionals
9	4	30	controls
10	17	29	data usage
11	17	36	data usage
12	9	25	data usage
13	22	26	calculations
14	0	34	conditionals
15	5	38	calculations
16	2	13	data usage

Table 2: Defect density

In summary, the majority of the errors (over 40%) were due to the use of "magic numbers." The average defect density was 10 and can be used as an initial benchmark for the amount of errors on any given page prior to conducting the inspection. Further analysis

of the defects metrics produced a strongly significant association between data manipulation and program control, with the majority of defects discovered in the former. The information obtained from this analysis can be utilized in future inspections to guide inspectors on areas of concentration, making changes to the checklist, improving the inspection process, and improving the software development process of NPSNET. The metrics collected as a result of this inspection are only a fraction of the NPSNET code. The findings tend to support the utility of code inspections in the validation of virtual reality systems.

Test	Group	Instances	Mean	Standard Deviation
Code Lines	1-19	2	1.0	1.41
"	20-29	7	13.1	4.78
"	30+	7	9.4	8.00
"	Total	16	10.0	7.08
Defects Content	Declaration	7	10.4	6.05
"	Data Usage	6	13.8	6.11
"	Conditional	1	9.0	0.00
"	Total	14	11.8	5.90
Page Content	Data	10	13.2	6.46
"	Control	5	5.6	4.39
"	Total	15	10.7	6.79

Table 3: Statistical Analysis

3. Observations

After conducting the trial inspection, the inspection process and checklist were evaluated with two members of the NPSNET research group. First, each member was

presented with a copy of the results of the inspection and were asked to look over the items in the checklist that did not result in a defect being detected. As a result of the discussion, categories 9, 16, 27, 28 and 29 were removed from the checklist, due to limited use or non-use. New checklist items were added based on the findings in the inspection of issues raised which did not result from the inspection checklist. For example, a non-checklist issue raised was that obsolete code was left in files to document how things were done previously or as a design decision on what variables could be added to the functionality of the module. It was decided it should have been properly documented in the code and this issue was added to the checklist.

Lastly, we discussed NPSNET in general, beginning with the limited use of comments. It was brought to my attention that in the past operability was primary and understanding was secondary in code generation. As NPSNET expanded, this approach has been counterproductive in the area of expansion and maintainability of the code. However, with the new style guide addressing the issue of comments in the code, the members are confident that the overall expansion and maintainability of the code will improve. The use of inconsistent style in representing floating point numbers is a result of no standards being set at time of code generation. These issues should be resolved with the use of the checklist.

The final question discussed with the members had to do with the primary language used, C or C++. In the code reviewed, the majority of the code resembled ANSI C with C++ used intermittently. The predominant language to date is C++, but NPSNET was originally written in C and over time C++ became the language of choice. Older modules have not undergone a complete revision to the C++ language constructs.

IV. SUMMARY AND CONCLUSIONS

The use of software inspections has seen enormous growth during the last decade as more studies reveal the benefits in verification and validation of software from their use [Ref. 14]. As the need and use of virtual reality systems increase, so does the need for reliable results from such systems. This thesis demonstrates how an inspection process can be defined for a virtual reality system to help satisfy the need for reliable results.

A. RESEARCH SUMMARY

This thesis presents research that developed a two-person code inspection for NPSNET. The purpose was two-fold: to improve the validation process and to increase productivity in coding. The approach chosen was based on the inspection method presented by Michael E. Fagan, developed for IBM and published in 1974 [Ref. 15]. The development of the inspection process began with the development of the checklist. The checklist was based on previous research and results from this thesis. The design of the inspection process focused on reducing the preparation time and the number of participants to conduct an inspection by having only two knowledgable members participate in the inspection. This thesis demonstrates that it is possible to develop an inspection process that minimizes the amount of time and people used to conduct inspections, and still have the inspection sensitive enough to locate coding problems.

The research group as a whole welcomed the results of the trial inspection and use of the checklist in general. They were confident from the results of the trial inspection that code examined from its use would improve the performance, operability, maintainability, and expansion of NPSNET.

B. APPLICATION

Although the code inspection was developed for two participants, its application is easily extendable to allow more participants. The flexibility is provided by the individual inspection portion of the inspection process. This affords the luxury of adding additional

inspectors without major modification to the inspection process. The individual inspection portion also allows the inspection leader to assign inspectors different sections of the code to inspect, increasing the code coverage.

Other NPS research groups and research groups in general may find the NPSNET code inspection useful as it is effectively a case study of how to implement software inspections. The main difference between the NPSNET inspection process and the others would be the inspection materials utilized by the various research groups as these are tailored to the specific projects.

The NPSNET code inspection process might also be of interest to software developers interested in instituting software inspections at their facility. By utilizing the two-person inspection process initially, one can implement the inspection process on an experimental basis. After establishing a history of its benefit, based on local experience, one can present the results to management for establishment of policy.

C. FUTURE RESEARCH

The NPSNET research group is currently in the process of re-designing their software development process. This thesis dealt with the coding phase of the software development process. As a result, the need for development of inspection materials dealing with the design phase was demonstrated. With the majority of NPSNET code uninspected by a formal inspection process, further inspections need to be conducted to aid in the validation of the system. This entails the collection of additional metrics and further statistical analysis to improve the inspection process in the future.

With the recent introduction of configuration management to the NPSNET system, the research group should support error collection. The evaluation of errors detected by other means, after the software inspection has been conducted, needs to be incorporated into the inspection checklist to aid in early detection and provide some overlap between the differing methods of testing.

In addition, an automated tool to assist program developers in ensuring compliance of their code to the rules of the style guide needs to be developed. Such a tool will decrease errors and provide inspectors more time to search for major defects during the inspection. Other automated tools to examine parser-determinable inspection questions (for example, 1A, 2A and 6A) need to be developed to assist the software development process and reduce the amount of time addressing these issues.

APPENDIX A. INITIAL NPSNET CODE INSPECTION CHECKLIST

1. Data Declaration

A. Default attributes understood (i.e., no lazy declarations)?

B. Correct lengths, types, and storage classes assigned?

C. Initialization consistent with storage class?

D. Any variables with similar names and dissimilar purpose?

2. Arrays

A. Is the array not dimensioned to a hard-coded constant (magic number)?

B. Is the array initialized and consistent with use?

3. Constants

A. Does the value of the variable change?

B. Are constants not declared with the preprocessor **#define** mechanism?

4. Scalar Variables

A. Does a negative value of the variable make sense? If not, is the variable *unsigned*?

B. Does the code explicitly declare **char** as either *signed* or *unsigned*?

C. Does the program use **int** or **float** to enhance running time (IRIS Performer)?

5. Classes

A. Does the class have any virtual functions? If so, is the destructor virtual?

B. Does the class have all of the following: Copy-constructor, Assignment operator or Destructor?

6. Data Reference

A. No unset variables used?

B. Subscripts within bounds?

C. Subscripts have integer values?

D. No dangling references? ***** Moved to RETURN VALUES *****

7. Strings

A. Is the string initialized properly?

B. Space declared for null-termination?

C. Is the string null-terminated?

D. String limits not exceeded?

8. Buffers

A. Are there always size checks when copying into the buffer?

B. Is the buffer large enough to hold its contents?

9. Bitfields ***** Removed *****

A. Is a bitfield really required for this application?

B. Are there possible ordering problems (portability)?

10. Local Variables

A. Are local variables initialized before being used?

B. Are C++ locals created and then initialized when needed?

11. Macros (inline functions)

A. Are macros not declared with the preprocessor **#define** mechanism?

12. Allocating Data

A. Is enough space being allocated?

B. Is **new** used in lieu of **malloc()**, **calloc()**, or **realloc()**?

13. Deallocating Data

A. No arrays are deleted as if they were scalars?

B. Is **delete** used in lieu of **free**?

14. Pointers

A. When dereferenced, can the pointer ever be NULL?

B. When copying dynamic memory, one copies the complete structure and not just allocate a copy of what the first pointer points to?

15. Casting

A. The code does not rely on implicit type conversion? ***** Modified *****

16. Files ***** Removed *****

A. Files opened before use?

B. End-of-file conditions handled?

C. File attributes correct?

D. Is the FILE argument of fprintf included?

E. Is a temporary file name unique?

F. Is a file pointer reused after closing the previous file?

G. Is a file closed in case of an error return?

17. Computation

A. No mixed-mode computations?

B. No division by zero?

C. Operator precedence understood? ***** Modified *****

18. Comparison

A. Comparison relationships correct (e.g., <, >, =, !=, etc.)?

B. Boolean expressions correct (e.g., &&, ||, etc.)?

C. Comparison and boolean expressions mixed correctly?

D. Operator precedence understood (parenthesized)?

E. Compiler evaluation of boolean expressions understood (short circuit)?

19. Conditionals

A. Are exact equality tests not used on floating point numbers?

B. Are signed variables not tested for equality to zero or another constant?

C. If the test is an error check, is the "error condition" not legitimate in other cases?

20. Control Flow

A. Will loop terminate?

B. Any loop bypasses because of entry conditions not set?

C. No off-by-one iteration errors?

21. Control Variables

A. Is the lower limit an inclusive limit?

B. Is the upper limit an exclusive limit?

22. Branching

A. In a switch statement, is every case terminated with a **break** statement?

B. Does the switch statement have a **default** branch?

23. Interfaces

A. All references to parameters associated with current point of entry?

B. Global variable definition consistent across modules?

24. Assignment Operator

A. Does "a += b" have the same meaning as "a = a + b"? ***** Modified *****

B. Is the argument for a copy constructor or assignment operator **const**?

C. Does the assignment operator test for self-assignment?

D. Does the assignment operator return a **const** reference to **this**?

25. Argument Passing

A. Are non-intrinsic type arguments passed by reference?

B. No hard-coded constants passed as arguments (magic numbers)?

C. All input-only arguments declared **const**?

26. Return Values

A. Is the return value of a function call being stored in a type that maintains precision?

B. Does a public member function return a **const** reference or pointer to member data?

C. Does a public member function return a **const** reference or pointer to data outside the object?

D. Does an operator return an object when it should, instead of a reference?

E. Are objects returned by **const** references?

27. Input/Output ***** Removed *****

A. OPEN statements correct?

B. Format specification matches I/O statement?

C. Buffer size matches record size?

D. I/O errors handled?

E. No textual errors in output information?

28. General Functions ***** Removed *****

A. Is this the correct standard function call (i.e., strchr instead of strrchr)?

B. Can this function not violate the preconditions of a called function?

29. Varargs Functions ***** Removed *****

A. Are there no extra arguments?

B. Do the argument types explicitly match the conversion specifications in the format string?

APPENDIX B. NPSNET CODE INSPECTION CHECKLIST

1. Data Declaration

A. Default attributes understood (i.e., no lazy declarations)?

```
    const float AspectRatio = 1.653;     // Default of literal promotes to double.
should be:
    const float AspectRatio = 1.653f;    // Enhances performance of IRIS Performer.
```

B. Correct lengths, types, and storage classes (e.g., **const**, **static**, **virtual**, etc.) assigned?

C. Any variables with similar names and dissimilar purpose?

2. Arrays

A. Is the array not dimensioned to a hard-coded constant (magic number)?

```
    int DaysInMonth[12];
should be:
    int DaysInMonth[MonthsInYear];
```

B. Is the array initialized and consistent with use?

C. No off-by-one iteration errors in indexing?

D. Is the array dimensioned properly?

```
    const unsigned int MaxColors = 32;
    const unsigned int MaxFlamePlumes = 5;
    unsigned int FlamePlumes[MaxColors];
should be:
    unsigned int FlamePlumes[MaxFlamePlumes];
```

3. Constants

A. Does the value of the variable change?

```
int MonthsInYear = 12;
```

should be:

```
const unsigned int MonthsInYear = 12;
```

B. Are constants not declared with the preprocessor **#define** mechanism?

```
#define MaxFlamePlumes 5
```

should be:

```
const unsigned int MaxFlamePlumes = 5;
```

C. Do all constant literals explicitly declare their type suffix (e.g., 1.525F,40000L, etc.)?

4. Scalar Variables

A. Does a negative value of the variable make sense? If not, is the variable *unsigned*?

```
int age;
```

should be:

```
unsigned int age;
```

B. Does the code explicitly declare **char** as either *signed* or *unsigned*?

```
typedef char SmallInt;
SmallInt Temp = 280;    // WRONG on Borland C++ 3.1
                        // or MSC/C++ 7.0!
```

should be:

```
typedef unsigned char uSmallInt;
typedef signed   char SmallInt;
```

C. Does the program use **int** or **float**, when necessary, to enhance running time (IRIS Performer)?

To take advantage of IRIS Performer's math package avoid the use of **short int**s and prefer single precision to double precision in all calculations.

5. Classes

A. Does the class have any virtual functions? If so, is the destructor virtual?

Classes having virtual functions should always have a virtual destructor. This is necessary since it is likely that one will hold an object of a class with a pointer of a lesser derived type. Making the destructor virtual ensures that the right code will be executed if the object is deleted via the pointer.

B. Does the class have all of the following: Copy-constructor, Assignment operator or Destructor?

If not, in general, one will need all three. The exception may be found for classes having only a destructor without the other two.

6. Data Reference

A. No unset variables used?

B. Subscripts within bounds?

C. Subscripts have integer values?

7. Strings

A. Is the string initialized properly?

B. Space declared for null-termination?

C. Is the string null-terminated?

D. String limits not exceeded?

8. Buffers

A. Are there always size checks when copying into the buffer?

B. Is the buffer large enough to hold its contents?

9. Local Variables

A. Are local variables initialized before being used?

10. Macros (inline functions)

A. Are macros not declared with the preprocessor **#define** mechanism?

```
    #define Max(A,B) ( (A) > (B) ? (A) : (B) )
```

should be:

```
    inline int Max(int A, int B) { return A>B ? A : B; }
```

not quite the same as the macro above, because this version can only be called with **int**s, however, templates solve that detail:

```
    template<class T>
    inline T& Max(T& A, T& B) { return A>B ? A : B; }
```

if templates not supported then

```
    #define GenerateMax(T) inline T& Max(T& A, T& B) { return A>B ? A : B; }
```

11. Allocating Data

A. Is enough space being allocated?

B. Is **new** used in lieu of **malloc()**, **calloc()**, or **realloc()**?

The problem with **malloc()**, **calloc()**, or **realloc()** is that they know nothing about constructors or destructors and use with objects that have constructors results in unexpected program behavior.

12. Deallocating Data

A. No arrays are deleted as if they were scalars?

```
    delete MyCharArray;
```

should be:

```
    delete [] MyCharArray;
```

B. Is **delete** used in lieu of **free**?

The problem with **free** is that it knows nothing about constructors or destructors and use with objects that have destructors results in unexpected program behavior.

13. Pointers

A. When dereferenced, can the pointer ever be NULL?

B. When copying dynamic memory, one copies the complete structure and not just allocate a copy of what the first pointer points to?

14. Casting

A. The code does not rely on an implicit type conversion (where inappropriate)?

15. Computation

A. No mixed-mode computations?

B. No division by zero?

C. Is operator precedence explicit (parenthesized)?

```
    Temp = 9-5+3    // What value of Temp (7 or 1)?
should be:
    Temp = (9-5)+3; // Temp is 7.
    Temp = 9-(5+3); // Temp is 1.
```

16. Comparison

A. Comparison relationships correct (e.g., <, >, =, !=, etc.)?

B. Boolean expressions correct (e.g., &&, ||, etc.)?

C. Comparison and boolean expressions mixed correctly?

D. Operator precedence understood (parenthesized)?

E. Compiler evaluation of boolean expressions understood (short circuit)?

17. Conditionals

A. Are exact equality tests not used on floating point numbers?

B. Are signed variables not tested for equality to zero or another constant?

C. If the test is an error check, is the "error condition" not legitimate in other cases?

18. Control Flow

A. Will loop terminate?

B. Any loop bypasses because of entry conditions not set?

C. No off-by-one iteration errors?

19. Control Variables

A. Is the lower limit an inclusive limit?

B. Is the upper limit an exclusive limit?

> A whole class of off-by-one errors are eliminated by using inclusive lower limits and exclusive upper limits.

20. Branching

A. In a switch statement, is every case terminated with a **break** statement?

B. Does the switch statement have a **default** branch?

21. Interfaces

A. All references to parameters associated with current point of entry?

B. Global variable definition consistent across modules?

22. Assignment Operator

A. Are user-defined operator overloading consistent with C++ built-in operators?

In other words, does "a += b" have the same meaning as "a = a + b"?
Programmers should never change the semantics of relationships between intrinsic operators.

B. Is the argument for a copy constructor or assignment operator **const**?

C. Does the assignment operator test for self-assignment?

operator=() should always start out with: if (this == &RightHandSide) return *this;

D. Does the assignment operator return a **const** reference to **this**?

This convention allows the user to write legal C++.

23. Argument Passing

A. Are non-intrinsic type arguments passed by reference?

```
MyType& MyFunction(MyType MyParameter);
```
should be:
```
MyType& MyFunction(const MyType& MyParameter);
```

B. No hard-coded constants passed as arguments (magic numbers)?

C. All input-only arguments declared **const**?

24. Return Values

A. Is the return value of a function call being stored in a type that maintains precision?

```
float& AreaOfCircle( const double& Radius );
```
should be:
```
double& AreaOfCircle( const double& Radius );
```

B. Does a public member function return a **const** reference or pointer to member data?

C. Does a public member function return a **const** reference or pointer to data outside the object?

D. Does an operator return an object when it should, instead of a reference?

E. Are objects returned by **const** references?

F. No functions return a reference to a local variable (dangling reference)?

25. Miscellaneous Code

A. No hard-coded constants (magic numbers) in code?

B. No commented out code in the module?

C. Do all floating point constants have at least one digit before and after the decimal point (e.g., 1.0F)?

Style Guide Section

26. Naming

A. Do all variables have meaningful names?

B. Are all identifier names in lower or mixed case (e.g., function, typedef, variable names, class, struct, union, and enum tag names)?

C. Are enum elements all in CAPS, but instantiations can be lowercase?

D. Do all names for a class end in "Class", typedef in "Type", and enum in "Enum"?

```
class car {...
};

typedef car auto;
enum autoMake {TOYOTA, HONDA, FORD, CHEVY};
```

should be:

```
class carClass {...
};

typedef carClass autoType;
enum autoMakeEnum {TOYOTA, HONDA, FORD, CHEVY};
```

E. Are names with a leading or trailing underscore used only on preprocessor **#defines**?

F. Are **#define** and **const** names all in CAPS?

G. Do all global variable names start with "G_"?

H. Do all global variable names, used in a single file, start with "L_"?

Remember: **#define** and **const** names are not global variables.

27. Indentation

A. Is all code indentation based on **three spaces** vice tabs?

B. Does indentation conform to "Style Guide"?

```
//This function is an example of the indentation requirements.
void func(type argument)
{
    //body of function
    if (expression){  //The use of braces is required for all control structures.
        //Do something.
        statement(s);
    }
    else if{ //Clause not required.
        statement(s);
    }
    else{ //Clause not required.
        statement(s);
    } //end if

    for (expression; expression; expression){
       statement(s);
    }

    do{
        statement(s);
    } while (expression);

    while (expression){
        statement(s);
    }

    switch(expression){
        case constant: //The first case
            //Do something.
            statement(s);
            break;  //Always have a break statement after each case.
        default:    //Always have a default case.
            statement(s);
            break;
    } //end switch
} //end func
```

C. Do all preprocessor commands (#xxxxx) start in the first column?

D. Are all functions separated from other text/code by **at least two blank lines**?

28. Classes

A. Are class functions and member data declared in the proper order?

```
//This class is an example of the proper order.
class nameClass {
    //Friends.
    friend class friendClass;
    friend int functName(int);

public:
    nameClass();    //Constructor.
    ~nameClass(); //Destructor.
    void publicFunct();
    int publicData;

protected:
    void protectedFunct();
    int protectedData;

private:
    void privateFunct();
    int privateData;
} //end nameClass.
```

Friend functions and classes should be used sparingly and only under very rare cases.

29. Functions

A. Does every function have a prototype?

B. Does the function have one **return** statement?

C. Does the main() function return an **int** and has a valid **return** statement included?

D. Is the return value of a function tested?

30. Globals

A. Is a shared variable between multiple files conform to the "Style Guide"?

The main source file has this construct:

```
#define _COMMON_
```

and the global shared header file has:

```
#ifndef _COMMON_
#define _COMMON_ extern
#endif
```

B. Are globals variables only declared in the global header file?

should be:

```
_COMMON_
int G_addCounter; //A counter used to keep track of the number of additions.
```

31. Files

A. Do all source files have the proper extensions, C++ is ".cc", C is ".c", and all headers are ".h"?

B. Do all source files have an associated header file, except the source file that has the main function included?

C. Does the source file include only the necessary header files (no extras)?

32. Code Files

A. Does a file that contains class member functions **only** contain class member functions?

B. Does the source code file contain the proper elements?

```
//This is an example of the source file format.
- File comment (see item 34B).
- All defines used to block out sections of code.
- Header (#include) files, standard library headers then user headers.
- Local Globals.
- Functions (Code).
//end of file source_file.cc
```

33. Header Files

A. Is there only one independent base class declared in the header file?

B. No variables are declared in the header file, except globals?

C. Does the header file contain the proper elements?

```
//This is an example of the header file format.
#ifndef _HEADER_FILE_H  //Always included with the name of the header file.
#define _HEADER_FILE_H
- File comment (see item 34B).
- Header (#include) files, standard library headers then user headers.
- #defines, enumerations or constants.
- User defined data types (typedefs).
- Globals.
- Function prototypes.
#endif
//end of file header_file.h
```

34. Comments

A. Are C++ "//" constructs used for all comments?

B. Do source and header files have a descriptive comment at the top?

```
// Name:
// Project:
// Operating Environment:
// Compiler:
// Description:
```

C. Does the function have a comment description?

```
// Function:
// Return Type: void
// Parameter: int count - the current program count value
//            float *value - the returned value of the static counter
// Purpose: This function takes a counter and converts it to the lapsed time

void myFunc(int count, float *value)
{
    *value = count / TIMER;
}
```

LIST OF REFERENCES

1. Gilb, Tom, Graham, Dorothy, "Software Inspection", *Addison-Wesley*, 1993.

2. Zyda, Michael J., Pratt, David R., Falby, John S., Barham, Paul T. and Kelleher, Kristen M., "NPSNET and the Naval Postgraduate School Graphics and Video Laboratory", *Presence*, Volume 2, No. 3, March 1994, pp. 244-258.

3. Torsiello, Kevin A., "Acoustic positioning of the NPS Autonomous Underwater Vehicle (AUV II) during hover conditions.", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

4. Book, S., "Improving Software Characteristics of a Real-time System Using Reengineering Techniques.", Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

5. Myers, Glenford J., "The Art of Software Testing", *John Wiley & Sons, Inc.*, 1979.

6. "IEEE Standard for Software Reviews and Audits", *IEEE Software Engineering Standards Collection*, Spring 1991, Std 1028-1999.

7. "Defense System Software Development", *Department of Defense*, 29 February 1988, DOD-STD-2167A.

8. "Technical Reviews and Audits for Systems, Equipments, and Computer Software", *Department of Defense*, 4 June 1985, MIL-STD-1521B.

9. Weller, Edward F., "Lessons from Three Years of Inspection Data", *IEEE Software*, September 1993.

10. Baldwin, John T., "An Abbreviated C++ Code Inspection Checklist", 27 October 1992.

11. Meyers, Scott, "Effective C++", *Addison-Wesley*, 1992.

12. Falby, John, "Style Guide for Winter AY95", Lecture Notes, Department of Computer Science, Naval Postgraduate School, 3 January 1995.

13. "IRIS Performer Programming Guide", *Silicon Graphics, Inc.*, 1992.

14. Ackerman, A. Frank, Buchwald, Lynne S. and Lewski, Frank H., "Software Inspections: An Effective Verification Process", *IEEE Software*, May 1989.

15. Fagan, M. E., "Design and code inspections to reduce errors in program development", *IBM Systems Journal*, Volume 15, No. 3, 1976, pp. 182-211.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 52 2
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Professor Timothy J. Shimeall, Code CS/Sm 4
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Instructor John S. Falby, Code CS/Fa 4
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Professor Michael J. Zyda, Code CS/Zk 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

7. Professor David R. Pratt, Code CS/Pr 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

8. Paul T. Barham..... 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

9. LT Charles E. Adams..... 2
   324 English Avenue
   Monterey, CA 93940